Overview

- Encapsulation
- Inheritance
- Polymorphism
- Abstract classes
- Interfaces

Access Modifiers

- `public` – can be accessed from anywhere
- `private` – can be accessed only within the class
- `protected` – can be accessed within the class, in subclasses, or within the same package
- No modifier – can be accessed within the same package

Access Control Example

```java
public class Foo {
    public int a;
    private int b;
    protected int c;
    public Foo() {
        a = 1;
        b = 2;
        c = 3;
    }
}

public class Bar {
    public Bar () {}
    public void print( Foo f ) {
        System.out.println(f.a); // ??
        System.out.println(f.b); // ??
        System.out.println(f.c); // ??
    }
    public static void main( String args[] ) {
        (new Bar()).print( new Foo() );
    }
}
```

Encapsulation

- Separate implementation details from interface
  - Control access to internal data
  - Change class implementation without breaking code that uses the class

Access to Private Fields

- `Getter` and `Setter` methods
  - `Point`  
    - `getX(), getY()`  
    - `setX(), setY()`  
  - What not just make x, y public??
Inheritance

- Code re-use
- **Subclass** inherits members of a **superclass**
  - Class variables
  - Methods
  - Except constructors
- Inherits != Can Access
  - public and protected
  - Subclass may have more members than the superclass

**Keyword** `super`

- A reference to the superclass
- A reference to a constructor of the superclass

**Keyword** `final`

- A final class cannot be inherited
  - public final class Classname {...}
- A final variable cannot change its value
  - Similar to `constants` in other languages
  - Convenience
  - Readability
  - `final double PI = 3.1415926;`

**Overriding**

- A subclass method has the same `signature` as a method of the superclass
- Method `signature`
  - Access modifier
  - Return Type
  - Name
  - List of parameters

**Overriding Examples**

- `public String toString()`
  - All Java classes implicitly inherit from the `Object` class
  - `toString()` is one of the methods defined in the `Object` class

**Polymorphism**

- An object of a subclass can be used as an object of the superclass
- The reverse is not true. Why??
Polymorphism Example

```java
public class A {
    public A() {} // A1
    public void afunc() {
        System.out.println( "afunc" );
    }
}

public class B extends A {
    public B() {} // B1
    public void bfunc() {
        System.out.println( "bfunc" );
    }
}

A a1 = new A(); // OK
B b1 = new B(); // OK
A a2 = new B(); // OK
B b2 = new A(); // Error!
```

Polymorphism Example

```java
public class A {
    public A() {} // A1
    public void afunc() {
        System.out.println( "afunc" );
    }
}

public class B extends A {
    public B() {} // B1
    public void bfunc() {
        System.out.println( "bfunc" );
    }
}

public class C extends B {
    public C() {} // C1
    public void cfunc() {
        System.out.println( "cfunc" );
    }
}

A a1 = new A(); // OK
B b1 = new B(); // OK
C c1 = new C(); // OK
A a2 = new B(); // OK
B b2 = new A(); // Error!
A a3 = new C(); // OK
```

Dynamic Dispatching

- When multiple implementations of the same method exist due to overriding, which method to invoke is determined by the actual class of the object.
- Dynamic means the decision is made at runtime (as oppose to compile time).

```java
Dynamic Dispatching Example

public class A {
    public A() {} // A1
    public void afunc() {
        System.out.println( "afunc" );
    }
}

public class B extends A {
    public B() {} // B1
    public void bfunc() {
        System.out.println( "bfunc" );
    }
}

public class C extends B {
    public C() {} // C1
    public void cfunc() {
        System.out.println( "cfunc" );
    }
}

A a1 = new A(); // OK
B b1 = new B(); // OK
C c1 = new C(); // OK
A a2 = new B(); // OK
B b2 = new A(); // Error!
A a3 = new C(); // OK
```

Shapes

- Attributes
  - Location
  - Length, width, Radius
- Operations
  - Move
  - Draw

Shape Class

```java
Shape Class Shape {
    protected int x, y; // initial location
    public Shape( int x, int y ) {
        this.x = x;
        this.y = y;
    }

    public void move( int newX, int newY ) {
        x = newX;
        y = newY;
    }

    public void draw() { ??? }
}
```
Abstract Shape Class

- An abstract class
  - Some operations are known and some are not
  - Unknown operations can be declared as abstract methods
  - Cannot be instantiated

```java
public abstract class Shape {
    int x, y; // location

    public Shape( int x, int y ) {
        this.x = x;
        this.y = y;
    }

    void move( int newX, int newY ) {
        x = newX;
        y = newY;
    }

    public abstract void draw();
}
```

Subclasses of Shape

- Point, Rectangle, and Circle
- A concrete class
- A subclass of an abstract superclass
- Must implement (override) the abstract methods
- Can be instantiated

Sort Integers

```java
public void sort( int a[] )
{
    int left = 0;
    while( left < a.length-1 ) {
        int index = left;
        for( int i=left ; i < a.length ; ++i )
            if( a[i] < a[index] ) index = i;

        // swap a[index] and a[left]
        int tmp = a[index];
        a[index] = a[left];
        a[left] = tmp;
        ++left;
    }
}
```

Sort Objects

- Any objects that has a lessThan() method
  ```java
  public abstract class Comparable {
      public Comparable() {} 
      public abstract boolean lessThan( ?? o );
  }
  ```

A More General Sort

```java
public void sort( Comparable a[] )
{
    int left = 0;
    while( left < a.length-1 ) {
        int index = left;
        for( int i=left ; i < a.length ; ++i )
            if( a[i].lessThan(a[index]) ) index = i;

        // swap a[index] and a[left]
        int tmp = a[index];
        a[index] = a[left];
        a[left] = tmp;
        ++left;
    }
}
```

The Need for Multiple Inheritance

- What if we want to sort an array of Point?
  - Inherit both Shape and Comparable?
The Problem of Multiple Inheritance

public class A {
    public int x;
    public void foobar() {
        ...
    }
}

public class B {
    public int x;
    public void foobar() {
        ...
    }
}

public class C extends A, B {
    ...
}

Which x or foobar() does C inherit?

Interface

Java’s answer to multiple inheritance
A interface only contains
- Method declarations
  - No method implementations
  - All methods are implicitly public and abstract
- Constants
  - All constants are implicitly public, static, and final

Interface Examples

public interface ActionListener
{
    public void actionPerformed(ActionEvent ae);
}

public interface AdjustmentListener
{
    public void adjustmentValueChanged(AdjustmentEvent e);
}

public interface MouseListener
{
    public void mousePressed();
    public void mouseClicked();
    public void mouseReleased();
    public void mouseEntered();
    public void mouseExited();
}

Comparable Interface

public interface Comparable {
    boolean lessThan( Object c );
}

Interface Usage

public class Point extends Shape implements Comparable {
    public Point( int x, int y ) { super(x,y); }
    public void draw() { ... }
    public boolean lessThan( Object o )
    {
        Point p = (Point) o; // cast to a Point for comparable
        // ...
    }
} // end of class Point

Abstract Class vs. Interface

Abstract class
- An incomplete class
- Class variables
- Constructors
- Methods and abstract methods
- extends
- Single inheritance
- Cannot be instantiated

Interface
- Not a class at all
- Only constants
- No constructors
- Only abstract methods (method declarations)
- implements
- Multiple implementation
- Cannot be instantiated